Practical Defenses Against Storage Jamming[*]

J. McDermott and J.Froscher
Center for High Assurance Computer Systems
Naval Research Laboratory
Washington, D.C. 20375-5537
{mcdermott, froscher}@itd.nrl.navy.mil

Introduction

Storage jamming [15] is malicious but surreptitious modification of stored data, to reduce its quality. The person initiating the storage jamming does not receive any direct benefit. Instead, the goal is more indirect, such as deteriorating the position of a competitor. We assume that a Trojan horse does the storage jamming, since the Trojan horse may access data that the attacker cannot. Manual storage jamming is possible, but in general much less effective.

We call values that should be stored *authentic values*. We call values stored by a jammer *bogus values*. A storage jamming attack diverges the state of the stored data from the authentic state. The attacker expects the bogus state will adversely affect the victim's performance of some real-world task. On the other hand, the attacker does not want the user to experience a catastrophic failure. The attacker expects that the victim will not detect the source of the problem but will continue to use the damaged data for a relatively long time. We make this more precise with the notion of *lifetime*. We define the lifetime of a storage jammer as the number of jams it can perform against a specific system before being discovered. The discovery does not necessarily have to be made on the system being jammed. The lifetime of a storage jammer is a function of the rate and extent of its jamming, the specific user population, and the seriousness of its impact on the real world.

The most promising targets for storage jamming are systems with complex stored data, the authenticity of which cannot be determined by inspection. This includes legacy systems, distribution and inventory systems, simulations, data warehouses, and command and control systems. Newer systems are not necessarily less promising as targets. The current trend is to build information systems out of rapidly developed special purpose applications based on low-cost shrink-wrapped general purpose software packages. A practical defense should be readily applicable to these kinds of systems, without introducing significant overhead.

Storage jamming is an interesting problem. All the attacker has to do is occasionally write a wrong but plausible answer, via the application program that generates authentic data. In principle, a Byzantine generals solution [20] will prevent storage jamming attacks, but Byzantine generals solutions do not scale well and are completely unworkable in many of the potential target systems. It would be difficult to integrate the necessary voting protocols and any digital signatures into shrink-

[*] This work was supported by ONR. Any opinions, conclusions, or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views, policies, or decisions of the Office of Naval Research or the Department of Defense.

Report Documentation Page	*Form Approved* *OMB No. 0704-0188*

1. REPORT DATE **1997**	2. REPORT TYPE	3. DATES COVERED **00-00-1997 to 00-00-1997**

4. TITLE AND SUBTITLE **Practical Defenses Against Storage Jamming**	5a. CONTRACT NUMBER
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Naval Research Laboratory,Center for High Assurance Computer Systems,4555 Overlook Avenue, SW,Washington,DC,20375**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**		**12**	

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std Z39-18

wrapped general-purpose software packages. In legacy systems and rapidly developed custom applications, it might not be possible to establish architectures suitable for Byzantine generals solutions. Extensive use of voting and cryptographic protocols would adversely affect the performance of many information systems.

Effective attacks can occur within a single access control boundary. The bogus values then propagate via authorized reads, whenever data is shared. Effective attacks can occur at a granularity too fine for practical audit mechanisms to detect. Fine granularity attacks can also avoid intrusion detection. For this reason, these techniques alone are not generally useful in preventing storage jamming. Conventional data integrity approaches such as checksums do not work because the attack can be made before the integrity mechanism is applied. The jamming software has access to the data-generating application logic so checksums can be applied to bogus data. Simple replication is also problematic. Jamming software can exploit application logic so that the bogus values are replicated. (We are going to show how a restricted form of replication can be exploited to detect storage jamming.)

There are several security-oriented data integrity approaches [2, 6, 18, 21, 22] that are potentially applicable to the prevention of storage jamming. The latest, most comprehensive work is the NCSC's technical report on data integrity [13]. McDermott and Goldschlag [15] show that these defenses are only effective when *all* software is known to be partially correct. Wiseman [22] also makes the case that all software that generates, stores, or transmits data must be partially correct, to protect against internal data integrity attacks. Correctness proofs do not scale well to large information systems. Furthermore, since the current trend is toward rapidly developed special purpose applications based on low-cost shrink-wrapped general purpose software packages, we conjecture that there is no practical way to verify all critical data originating software. Instead, we propose a special kind of detection as the best defense.

In this paper, we present two classes of defense based on detection. We assume that a detected jammer can be removed. (Removal may be extraordinarily difficult in some mission-critical legacy systems, but that is another story.) Here, we first describe our use of assignment and unbounded nondeterministic iteration to model storage jamming. Next we use our model to explain two important classes of storage jamming attacks: *external jamming* and *internal jamming*. We then describe two defenses and discuss how effective they are against each class of storage jamming. We finish with some conclusions and discussion of current and future work.

A Model of Storage Jamming

Since we are interested in stored data, we adopt the Unity model of computation [5]. We chose Unity rather than an event- or trace-based framework because we are interested in modeling storage of data values. Storage and retrieval of data can be modeled in trace-based frameworks, but Unity models it in a more direct and natural way.

To accommodate the access control and complex data portions of our model, we extend Unity logic in three ways. To model complex data, we add a **definition** section to define new data types. (Conventionally, new types are not introduced in Unity.) We also add notation to distinguish between private variables and public variables. To model access control, we extend Unity by adding the notion of process. We model a process as a Unity program with a *context*. The context of a process includes all the variables within its scope, i.e., its private variables and the public variables of all processes, plus two more variables. Every process has a natural number that is its unique *process id*, and a natural number that is its unique *subject* (in the sense of HRU, BLP, and TAM). We use the union operator to assemble two or more processes into a concurrent system. The union of distinct processes is the same as the union of programs $P_1 [] P_2$ in conventional Unity logic; the corresponding sections of P_2 are appended to P_1.

We use a nested relational algebra [12] to model complex data in our Unity programs. Objects *x* are built up out of the base (i.e., immutable atomic) object types: B (Booleans) and N (natural numbers). Our base objects supply the values used in our data storage objects. We build complex types with two constructors: Cartesian products of types *x* and *y*, denoted **tuple of** (*x*; *y*), and finite sets of type *x*, denoted **set of** *x*.

To show the effect of access control on storage jamming, we include a fixed finite access control matrix *P* [2,8,19]. The matrix *P* is included as part of every guard associated with a variable protected by access control. We do not model the protection of private variables. Guarded assignment takes place according to the normal rules of Unity logic. An expression associated with a true guard is assigned to the target variables. If a process with subject *s* uses an access-control protected variable *x* on the left side of an assignment, then the guards for that assignment must test for the *write* privilege in *P*[*s,x*]. Likewise, if variable *x* appears in one or more expressions on the right side of an assignment or in the guard, then the guard must contain a test for the *read* privilege in *P*[*s,x*].

Storage Jamming Attacks

There are many ways to classify storage jamming attacks. One distinction that is important to defense is the difference between *external jammers* and *internal jammers*. If the jammer only targets objects that have been accessed (e.g., opened) at the request of a user, then the jammer is an internal jammer. If the jammer targets any objects that are accessible in its current context, then the jammer is an external jammer. From an attacker's perspective, both external and internal jamming may be used by the same Trojan horse.

External Jamming

The following program is an example of a Trojan-horse external jammer that shows all details of program logic, context and access control available in our model. Functions *f0* and *f1* represent the ostensible features of the Trojan horse, that is the original program that the Trojan horse mimics (i.e., its *host*). Function *g* computes a plausible but bogus value. Function *rand* is a pseudo random number generator imported from elsewhere; it returns values in the range of the idealized storage array *d*. Our jammer performs the required access control checks against *P* and does not jam if the current user does not have *write* privilege to the randomly selected element of *d*.

Lines 2 and 3 define input commands as triples containing the requested operation, the operand as an index into the data store, and an input parameter. Lines 4-9 declare the input command *u*, data storage *d*, access control matrix *P*, subject *s0*, and process id *p0* of this process. Lines 10-12 establish private variables for the Trojan horse. The always section of lines 13-15 is used to define certain variables as functions of other variables, in this case security conditions for the guards. Variables defined in the always section may only be used on the right hand side of assignments or initializations. Lines 16-17 initialize the jammer's count, pick an initial random target, and set the value of the input *u* to force the application to wait for its user to give a command. As each command is executed, it is acknowledged by emptying *u*. The authentic function of the program is on lines 19-

20. The Trojan horse assignments are on lines 21-24 (the malicious assignment of $g(u.val)$ to the randomly selected element of d).

```
1  program external_jammer
2      define
3          command = tuple of ( op : ℕ;  obj : ℕ; val : ℕ)
4      declare public
5          u : command;                               // user input
6          d : array [0…k] of ℕ;                      // idealized storage
7          P : array [0…i, 0…j]of privilege_set;      // the access control matrix
8          s0 : ℕ;                                    //the subject of this process
9          p0 : ℕ;                                    //the process id of this process
10     declare private external_jammer
11         count : ℕ;                                 //jammer's hidden counter
12         t: ℕ                                       //index of jammer's random target
13     always
14             sec0 = write ∈ P[s0, d[u.obj]] ∧ read,write ∈ P[s0, u]
15         ||  sec1 = read ∈ P[s0, u]
16     initially
17         count = 0 || t = rand(SEED) || u = null
18     assign
19             d[u.obj], u  :=  f0(u.val), null     if  u.op = 0 ∧ u ≠ null  ∧ sec0
20                          ~   f1(u.val), null     if  u.op = 1 ∧ u ≠ null ∧ sec0
21         ||  count        :=  count+1             if  count < JAM ∧ u ≠ null ∧ sec1
22                          ~   0                   if  count ≥ JAM ∧ u ≠ null ∧ sec1
23         ||  t, d[t]      :=  rand(t), g(u.val)   if  count ≥ JAM ∧ u ≠ null ∧ sec1
24                                                      ∧ write ∈ P[s0,d[t]]
25 end // external_jammer
```

Line 23 models the fact that external jammers do not have to be closely coordinated with the actions of their host program. This makes them relatively easy to develop. Many external jammers are relatively easy to defend against. If the attacker is willing to work harder, an internal jammer may be a more effective strategy.

Internal Jammer

We can change our external jammer into a internal jammer by replacing lines 18-24 (the assign section) with lines 26-33, to read

```
26     assign
27             d[u.obj], u  :=  f0(u.val), null  if  u.op = 0 ∧ count < JAM ∧ u ≠ null
28                                                   ∧ sec0
29                          ~   f1(u.val), null  if  u.op = 1 ∧ count < JAM ∧ u ≠ null
30                                                   ∧ sec0
31                          ~   g(u.val), null   if  count ≥ JAM ∧ u ≠ null ∧ sec0
32         ||  count        :=  count+1          if  count < JAM ∧ u ≠ null ∧ sec1
33                          ~   0                if  count ≥ JAM ∧ u ≠ null ∧ sec1
```

Notice that the replacement of the concurrency separator || on line 23 with the case separator ~ in line 31 indicates that the jammer's target is now the same as the user requested operand. The internal jammer also has a more complicated guard arrangement. We cannot copy the guard structure of the external jammer with its overlapping cases, because we are assigning to the same variable that the user requested. Unity logic requires that the case structure of the guards on lines 27-31 must be disjoint or assign the same value to $d[u.obj]$. Most real implementations would have to make similar checks or experience race conditions.

As it stands above, our internal jammer is not very effective. To assign a bogus value to $d[u.obj]$, it must fail to perform the requested operation *f0* or *f1*. This may be far too easy to detect. There are several ways to avoid the problem; we will show one that is easy to explain.

Complex Data

One reason for the internal jammer's difficulty is that it is jamming atomic data. In most applications, data is complex rather than atomic. Suppose we add a type definition to our internal jammer

```
        complex_data = tuple of (x : N; y : N)
```

Then we replace line 6 with line 34, changing the declaration of the data store to be

```
34      d : array [0…k] of complex_data;
```

Let the authentic changes be to the first component $d[u.obj].x$ of the requested operand. We replace lines 27-30 with lines 35-36, thus

```
35 d[u.obj].x, u    :=  f0(u.val), null   if  u.op = 0 ∧ u ≠ null ∧ sec0
36                  ~   f1(u.val), null   if  u.op = 1 ∧ u ≠ null ∧ sec0
```

Our jammer can now target the second component *y* of the requested object and have the overlapping case structure of the external_jammer (replace line 31 with line 37)

```
37        ||   d[u.obj].y := g(u.val)   if  u.op = 0 ∧ count ≥ JAM ∧ u ≠ null ∧ sec0
```

This attack works because neither *f0*, *f1*, nor access control (nor audit) is performed on atomic objects. This allows an internal jammer to make the authentic change while inserting bogus data at the same time. A similar effect could be achieved by delaying the bogus update until the user directs the application to close or release the current object. We do not show it because the Unity logic for this is too involved for our present discussion.

Internal jamming of complex data significantly reduces the effectiveness of audit as a detection defense. Auditing changes at the granularity of an atomic object is prohibitively expensive and may not be possible. Intrusion detection techniques cannot be generally successful for the same reason. The effectiveness of intrusion detection against external jammers is implementation dependent.

Defenses

It is time to look at defenses that do not require partial correctness at all points along the path from original input to ultimate output. Instead, we only need to show the correctness of the actual defensive mechanisms themselves, a much simpler task.

Cryptography

One possible defense is cryptographic techniques. Successful defense through cryptography is more difficult than it appears at first glance, particularly against internal jammers. There are three plausible ways that cryptographic techniques could be used: digital signatures, threshold schemes, and encryption. Because the bogus values can be inserted prior to signature, digital signatures are no more effective against storage jamming than checksums. A similar problem holds for threshold schemes. Threshold schemes suppose that the data values are already known before the separate shadows are constructed. The jammer has access to data-generating application logic so it can supply bogus values as input to the threshold scheme. An even more important limitation is that threshold schemes do not scale up well. That is, they cannot be incorporated into all of the pertinent components of a large information system, for both integration and performance reasons. Encryption of the data plausibly prevents a jammer from reading or modifying it. NRL has already constructed a Trojan horse that jams an encrypted database without attacking its cryptographic system. The difficulty arises because 1) the jammer has access to the application logic and unencrypted data, 2) the jammer operates with the user's authorization, and 3) encryption of atomic objects is impractical. If the application encrypts stored data, then the Trojan horse can invoke its host's internal cryptographic routines against its chosen target. Furthermore, operating- or file-system cryptographic defenses may not be able to distinguish legitimate requests to decrypt and open files from those submitted by a Trojan horse. Finally, if the unit of encryption is not atomic, then a jammer may attack a component of a complex object that has been decrypted object-wise for authentic processing. Techniques such as replay [16] may allow an external jammer to bypass the cryptographic protection.

We must point out that a careful encryption defense will stop many external jammers. For this reason, we encourage researchers to investigate low-overhead cryptographic schemes or protocols that will defeat external jammers. In the next section, we will also see a technique that will defeat many internal jammers as well.

Detection Objects

A detection object is an abstract mechanism intended to detect malicious software that does not return the correct results. The detection objects are only accessed by a *detection process* that checks their stored values. Detection objects appear to be authentic data, but the only authentic changes made to detection objects are made by the detection process. The detection process changes the detection object from one precomputed state to another. If, between these state changes, the detection process checks the detection object and finds it not in its precomputed state, then it was probably modified by a storage jammer. Detection objects are discussed in greater detail by McDermott and Goldschlag [15].

Abstract detection objects satisfy two properties

1. *Indistinguishability*: To any jamming process, a detection object is indistinguishable from a storage object.
2. *Sensitivity*: The only authentic process that modifies the detection object is the detection process.

The implementation problem for a detection object defense is to preserve both indistinguishability (avoid counter-detection) and sensitivity (avoid false detections).

In our example external jammer, we could model this by declaring a variable *a* to hold the subject of the security administrator. We could add the counter-detection by modifying line 23 to append a check[1] for ownership of the target.

```
38 ||   t, d[t] :=  rand(t), g(u.val)    if  count ≥ JAM ∧ u ≠ null ∧write ∈  P[s0, d[t]]
39                                           ∧ own ∉ P[a,d[t]]
```

This kind of counterdetection may not even require inside knowledge of a particular system. If a proposed defense is known to always operate in this mode, a jamming program may be able to obtain a list of user names that correspond to the security administrator role.

Plausible Domains

One way that we deal with the problem of defining and preserving indistinguishability is to define detection objects as always being taken from a *plausible domain*. Recall the notation $H_i.state$ for the state of a program at the i^{th} step in execution history H. We will treat $H_i.state$ as a set of objects. We introduce a type function t which gives the type of any object. We then define the plausible domain of object a, with respect to set of states Q, denoted $pdom(a, Q)$. We define $pdom(a, Q)$ as the set of all objects of type $t(a)$ that appear in some state $H_i.state$ in the set of states Q, or $pdom(a, Q) = \{x \in t(a) \mid x \in H_i.state \wedge H_i.state \in Q\}$. We define the plausible domain of a detection object d at step i of an execution history $pdom(d, Q_i)$ using the set of all states from execution steps prior to step i, that is, using $Q_i = \{ H_j.state \mid 0 \leq j < i \}$. We conjecture that this definition, if preserved in the implementation of all[2] detection objects on a system, will reduce the jammer to either guessing (i.e. we need probabilistic models) or trying to inspect the internal state of detection processes. This latter approach gives away much of the jammer's unique advantage because it is now trying to obtain unauthorized accesses and break a cryptosystem, actions which it previously did not need to perform.

The Replay Defense

Internal jammers pose a serious problem for detection object defenses. If an application containing an internal jammer never accesses detection objects, then an internal jammer will never access a detection object either. On the other hand, if we allow applications to access detection objects, then the integrity of the application will be compromised, to say nothing of the loss of sensitivity in the detection objects.

One answer to this problem is to create a separate subsystem that contains nothing but detection objects. The detection objects are created by replaying actual user-generated application commands. The commands are recorded as they are executed against real application data. The recorded commands are saved in a script and played back against a separate set of detection objects. The separate detection objects are copies of real application data. The state of the application objects at the moment recording started is used as the initial state of the detection objects.

Before using the script, we want to validate its correctness to rule out the (rare) case where a jammer will always jam following a fixed pattern, and is present during the creation of the script. Notice that

[1] Unlike HRU or TAM, our model uses checks for the absence of rights because we are not concerned about tractability.

[2] This means that all objects in the context of a detection object must also be taken from a plausible domain.

having a jammer attempt this does not guarantee the jammer will escape detection. The jammer must remain “in phase” with the script it contaminated, a difficult problem. To see this, suppose we call the length of our detection script its “period” and there is a user who also runs the application/jammer. Suppose the jammer’s “period”, that is, the number of commands the jammer skips before jamming, exactly matches the detector’s “period”. Even if the jammer contaminated the detector’s script, its operation versus the detector is interleaved with the user’s commands and thus is unlikely to remain “in phase” with the detector’s script. A jammer can remain “in phase” with a detection script if it is stateless, that is, keeps no information about its past behaviour. Stateless jammers are not very effective because it is difficult for an attacker to predict how they will behave. Stateless jammers are also likely to be discovered by ordinary testing. So any jammer that can be guaranteed to remain “in phase” with a script is guaranteed to show up in an analysis or validation of that scripts execution.

Here is an example of a simple replay detection object defense. It runs with our jammer as *internal_jammer* [] *detector*. This program uses the variable *u* to send commands to the internal_jammer and waits for the acknowledgment that comes in *u*. The function *check* computes a checksum for an array of *complex_object*; it is imported from elsewhere. The array *sum* contains the correct checksum for each state resulting from applying the proper script command, starting from the initial state held in the array *initial_store*. Notice that the checksum is applied to the entire store *d* and not just the operand of the previous step. This checks for external jammers. The detector cycles through L+1 states, checking the current state and sending a new command to the application. If the check fails, the alarm is latched so that future actions of the script do not accidentally remove the incorrect values and cover up the jamming.

```
program detector
    declare public
        s1 : ℕ;                             //the subject of this process
        p1 : ℕ;                             //the process id of this process
    declare private detector
        script = array [0…L] of command;    //recorded commands
        sum = array [0…L] of ℕ;             //stored checksums for script
        initial_store = array [0…k] of complex_object;
        state : ℕ;
        alarm : 𝔹;                          //an alarm to latch the detection event
    initially
        state = 0 || alarm = F  ||  <∀i : 0 ≤ i ≤k :: d[i] := initial_state[i]>
    assign
            u, state   :=  script[state], state+1      if  u = null ∧ state < L
                                                           ∧ read, write ∈ P[s1, u]
                        ~  script[state], 0            if  u = null ∧ state ≥ L
                                                           ∧ read, write ∈ P[s1, u]
        ||  <∀i : 0 ≤ i ≤k :: d[i] := initial_store[i]> if  u = null ∧ state ≥ L
                                                           ∧ read ∈ P[s1, u]
        ||  alarm      :=  check(d) ≠ sum[state]       if  u = null ∧ alarm = F
                                                           ∧ read ∈ P[s1, u]
                                                           ∧ read ∈ P[s1, d]
end // detector
```

If we use the replay defense in a context that contains detection objects that are not accessed as well as objects that are accessed then it is effective against both external jammers and internal jammers. The effectiveness depends on preservation of indistinguishability in the script and context of the detection objects. It also depends on the protection that can be afforded to the detection process, to keep jammers from inspecting or tampering with it. Access control can be used to set up a protected subsystem. Encryption of the script and checksums might also be used, depending on the strength of the access control. The detection process also needs assurance that it does not contain malicious software itself.

It is important to notice that the replay defense does not need to be applied to all data in a system, at all times. A subset can be protected adequately by running a replay against it. Since storage jamming is a continuous attack, the replay defense does not need to run at all times. It most cases, significant protection can be achieved by running it intermittently, at the convenience of the users.

Replication Defense

The *replication defense* is an architecture-specific defense that is an attractive alternative to the more general replay defense, when replication is present for other reasons. Where the replay defense addresses the internal jammer problem by making all objects detection objects, the replication defense has no detection objects at all. Instead, replicas of application data are updated through *logical*[3] updates. The logical updates are performed by applications with distinct provenances. Our notion of distinct provenance is not the same as n-version programming; we are not trying to tolerate faults but to deny a single attacker easy access to multiple sites. The expectation is that we have now forced would-be attackers to compromise multiple (possibly heterogeneous) host programs. Attacks which are unable to compromise multiple host programs should quickly fail. As we will show, even if multiple host programs are compromised, storage jamming may still be detected.

The detection process is much simpler in the replication defense, except that it is now two or more processes, one at the primary site and one at each replica. Following changes to protected data, the process at the primary site computes a checksum and sends it to each replica site, along with the identification of the change. After the logical update is performed at the replica site, the detection process at the replica site computes its own checksum and compares it to the checksum transmitted by the primary site detection process. If there is disagreement, there is a problem. It is not even necessary for the detection process at the replica site to detect the change when it first occurs. Jamming must cause the replica and the primary copy to ultimately diverge, unless there are jammers at every site, *that coordinate their bogus updates to also have one-copy serializable histories*. A set of multiple jammers can only do this if they are either stateless or carry out an unauthorized communication protocol.

The replication defense can be effective against both external jammers and internal jammers. Indistinguishability comes for free. The challenges now become: 1) maintaining a distinct provenance for the protected applications at each site, and 2) preventing coordination between jammers at different sites if distinct provenance fails.

---

[3] Recall that when data is replicated, a change may be transmitted either as a value or as the text of the command that computes the new value. The former is called a *physical* update; the latter a logical update.

In theory a distinct provenance is possible. In practice, some software may have commonality. Some software will either have been developed with the same tools or be based on the same packages. This raises the question of Trojan-horse-writing Trojan horses [17]. Fortunately, a would-be attacker introducing a jammer via widely-used software faces a significant problem. The problem is that the jammer’s lifetime is now likely to be expended against systems other than the target. The attacker must now arrange to turn off the jammer in systems that are not targets or risk premature discovery of the attack.

The attacker’s difficulty in preserving one-copy serializability over all jams is significant. From a transaction management perspective, the cooperating jamming programs must overcome the local indirect conflict problem [23] and they must deal with failures (i.e., at some sites the bogus value may not be successfully inserted on the first try).

Differences in local use of the jammer’s host application can introduce indirect local conflicts that are out of the jammer’s control. Resolving indirect local conflicts requires the introduction of explicit communication between sites, complex program logic, and special data structures. The greatest difficulty for one-copy serializability is not in synchronizing the bogus values but in synchronizing the internal states of the various jammers. Local use of the jammer’s host application can put the various jammers into inconsistent states, just as the extra commands of the replay defense force a jammer out of step with a contaminated script.

It might seem that replication mechanisms developed for multilevel security might be useful to jammers in dealing with failures. Fortunately, the work of Kang, et al., [11] has shown that these require either some explicit communication between sites involving complex logic and special data structures or human intervention.

Replication has the added advantage that it provides a means to continue operations while under attack. At least one of the copies of the data will be authentic. Furthermore, the authentic copies can be used to recover from the attack, using an algorithm similar to the one designed by Amman, Jajodia, et al. [1].

Conclusions

The distinction between internal jammers and external jammers is an important one. Although they are harder to construct, internal jammers pose a serious problem for cryptographic and detection object defenses. One reason for this is that the internal jammer can use its host program’s logic and security privileges to avoid detection objects or encryption.

The replay defense can detect internal jammers. It does this by replaying authentic commands against a separate subsystem that contains only detection objects. In many cases it will not be necessary to verify the either the replay script or the host program, because jammers present during the creation of the script will have difficulty remaining synchronized with the contaminated script.

The replication defense uses no detection objects per se, but uses logical updates submitted to applications with distinct provenances. This requires attackers to compromise multiple host programs. Even if the attackers accomplish this, the replication defense will probably still detect a storage jamming attack because the multiple Trojan horses will not be able to maintain a one-copy serializable history over their bogus updates and internal states. The possibility of indirect local conflict or failure makes it unlikely that a storage jammer can remain active against a replication defense.

We are currently prototyping the replay defense as the Doc prototype and testing it against two prototype internal storage jammers Ike and Curly Bill. Our future plans include prototypes of the replication defense, as well as further modeling of the problem and related defenses. We would hope to see other researchers investigate two outstanding problems: 1) low-overhead cryptographic defenses, and 2) architectures that allow easy removal of malicious code.

References

1. AMMANN, P., JAJODIA, S., McCOLLUM, C., and BLAUSTEIN, B. Surviving information warfare attacks on databases. In *Proceedings of the IEEE Symposium on Security and Privacy*, (Oakland, California, 1997), 164-174.

2. BELL. D. and LA PADULA, L. Secure Computer Systems: Unified Exposition and Multics Interpretation. MTR-2997, MITRE Corp, 1975.

3. BOEBERT, W.E. and KAIN, R.Y. A practical alternative to hierarchical integrity policies. In *Proceedings of the 8th National Computer Security Conference* (Gaithersburg, Maryland, 1985). 18-28

4. BREITBART, Y., GARCIA-MOLINA, H. and SILBERSCHATZ, A. Overview of multidatabase transaction management. *VLDB Journal*, 1,2 (October 1992) 181-240.

5. CHANDY, K.M. and MISRA, J. *Parallel Program Design: A Foundation*. Addison-Wesley, 1988.

6. CLARK, D.D. and WILSON, D.R. A comparison of commercial and military computer security policies. In *Proceedings of the IEEE Symposium on Security and Privacy* (Oakland, California, 1987).

7. GEORGAKOPOLOUS, D., RUSINKIEWICZ, M., and SHETH, A. On serializability of multidatabase transactions through forced local conflicts. In *Proceedings of the 7th International Conference on Data Engineering* (Kobe, Japan, 1991).

8. HARRISON, M., RUZZO, W. and ULLMAN, J. Protection in operating systems. *CACM*, 19, 8, 1976, 461-471.

9. JAJODIA, S. and KOGAN, B. Transaction processing in multilevel-secure databases using replicated architecture. In *Proceedings of the IEEE Symposium on Security and Privacy*, (Oakland, California, 1990).

10. KANG, M. COSTICH, O. and FROSCHER, J. A practical transaction model and untrusted transaction manager for a multilevel-secure database system. In *Proceedings of the 6th IFIP Working Conference on Database Security*, (Vancouver, 1992), 289-310.

11. KANG, M., MOSKOWITZ, I., and LEE, D. A network pump. *IEEE Transactions on Software Engineering*, 22, 5 (May 1996), 329-338.

12. KORTH, H. and SILBERSCHATZ, A. *Database System Concepts*, McGraw-Hill, 1996.

13. National Computer Security Center. *Integrity in Automated Systems*, C Technical Report 76-91. September 1991.

14. MCDERMOTT, J. JAJODIA, S. and SANDHU, R. A single-level scheduler for the replicated architecture for multilevel-secure databases. In *Proceedings of the 7th Annual Computer Security Applications Conference*, (San Antonio, Texas, 1991).

15. MCDERMOTT, J. and GOLDSCHLAG, D. Storage jamming. In *Database Security IX: Status and Prospects* (D. SPOONER, S. DEMURJIAN, and J. DOBSON, eds.) Chapman and Hall, 1996.

16. MCDERMOTT, J. and GOLDSCHLAG, D. Towards a model of storage jamming. In *Proceedings of the 9th Computer Security Foundations Workshop* (County Kerry, Ireland, 1996).

17. McDERMOTT, J. A technique for removing an important class of Trojan horses from high order languages. In *Proceedings of the 11th National Computer Security Conference* (Baltimore, 1988), 114-117.

18. SANDHU, R. Separation of duties in computerized information systems. In *Database Security IV: Status and Prospects* (S. JAJODIA and C. LANDWEHR, eds.) North-Holland, 1991, 179-189.

19. SANDU, R. The typed access matrix model. In *Proceedings of the IEEE Symposium on Security and Privacy* (Oakland, California, 1992).

20. SILBERSCHATZ, A. and GALVIN, P. *Operating System Concepts*, 4th ed., Addison-Wesley, 1995, ISBN 0-201-50480-4.

21. THOMSEN, D. and HAIGH, T. A comparison of type enforcement and Unix setuid implementation of well-formed transactions. In *Proceedings of the Sixth Annual Computer Security Applications Conference* (Tucson, Arizon, 1990), 304-312.

22. WISEMAN, S. TERRY, P. WOOD, A. and HAROLD, C. The trusted path between SMITE and the user. In *Proceedings of the IEEE Symposium on Security and Privacy* (Oakland, California, 1988), 147-155.

23. ZHANG, A. and ELMAGARMID, A. A theory of global concurrency control in multidatabase systems. VLDB Journal, 2,3 (July 1993), 331-360.